Power Matters.

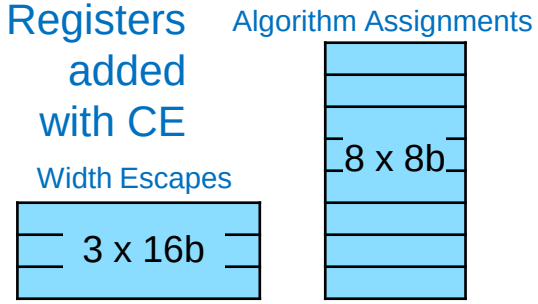
Using Proposed Vector and Crypto Extensions For Fast and Secure Boot

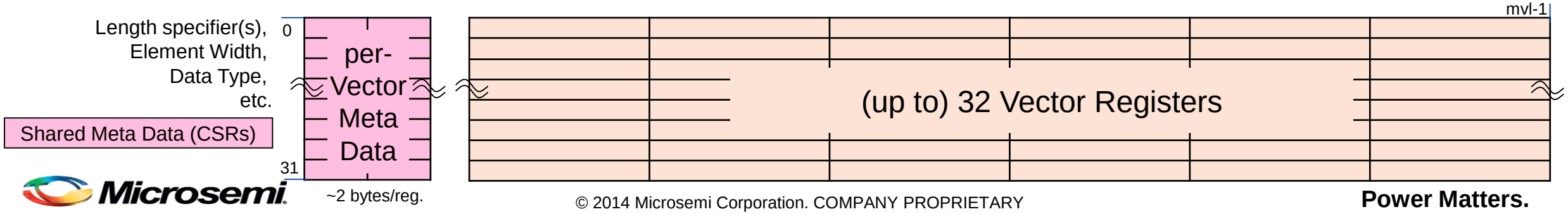
Several case studies in the use of the proposed cryptographic ISA extensions

G. Richard Newell from Microsemi Corp. and the RISC-V Foundation Security Working Group
With Derek Atkins, Drake Smith, and Michael Caiafa from SecureRF Corp.
For the 2nd day of the 7th RISC-V Workshop: Nov. 29, 2017
Sponsored by Western Digital in Milpitas, CA

Summary of Proposed RISC-V Cryptographic Extensions

- All crypto [using the extensions] will be done using the vector registers:
  - Public key algorithms will use SW with existing and new vector extension commands
    - Data widths will accommodate all the popular asymmetric algorithms and protocols (very big words!)
    - These work with existing vector instructions ***vmul***, ***vadd***; plus instructions are added for use with the Galois Field types that also include an extra register for the field-reduction prime or polynomial (***vmulr***, ***vaddr***)
  - Symmetric algorithms and digests will use a new vector opcode ***vcrypt***
    - The algorithm and function are encoded into the vector instruction: ***vcrypt.aes.enc*** ***vcipher, vplain, vkey***
  - Vector commands work on one element or multiple elements (1 ≤ **vl** ≤ **mvl**) per instruction issue
    - The implementation details: size/area tradeoff, single/multiple lanes, side-channel countermeasures, etc. are left up to the implementer
  - If strip-mined properly, object code will be portable to all RISC-V implementations with the same feature set, whether small/slow or large/fast
  - Profiles will be used to help direct implementations to standard feature sets
  - Still a work in progress… subject to change





Galois Counter Mode Using the Proposed Crypto Extensions

Initialization Vector (IV)

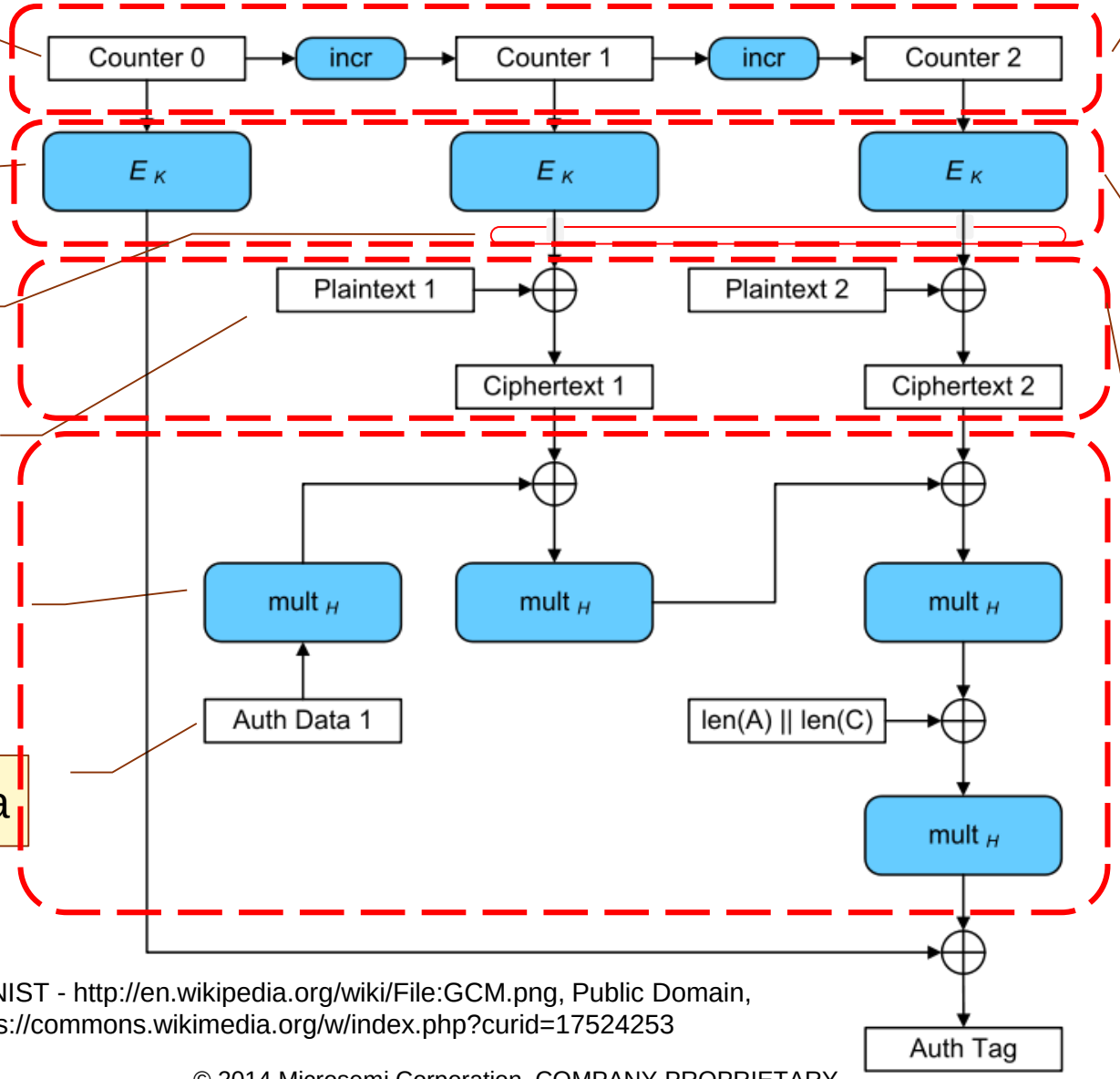
Encryption Algorithm (e.g. AES)

Keystream

Encrypted-and-Authenticated Data

Multiplication over $GF_{2^{128}}$

Authenticated-only Data



By NIST - http://en.wikipedia.org/wiki/File:GCM.png, Public Domain, https://commons.wikimedia.org/w/index.php?curid=17524253

vadd vctr, vctr, vl
Each time through strip-mine loop, add vl to all counter values

vcrypt.aes.enc vks, vctr, vkey
Encrypt counters to get keystream

vld vpt, a1
vxor vct, vks, vpt
vst vct, a2
XOR keystream with plaintext to get ciphertext (Increment a1 and a2)

This looks like it has serial dependencies, but it can be vectorized because GF multiplication is associative (See next slide)

Repeat until entire message is consumed

Vectorizing GHASH

- GHASH (iterative description)

$X_{i+1} = (X_i \oplus M_i)\bullet H$

$= (\ldots(((M_0\bullet H \oplus M_1)\bullet H \oplus M_2)\bullet H \oplus M_3)\bullet H\ldots)\bullet H$

i = 0 .. N-1
$X_0 = 0$

Where "•" is multiplication in $\mathbb{GF}_{2^{128}}$ with the reduction polynomial:
$g = x^{128} + x^7 + x^2 + x + 1$ (preload a register with binary of this)
vmulr does this operation if the data type is set to 'GP128'

$\oplus$ is addition in $\mathbb{GF}_{2^{128}}$ and is the same as XOR for any "g"

- GHASH (4-element vector description)

Easily extendable to more than 4 elements

$= (((M_0\bullet H^4 \oplus M_4)\bullet H^4 \oplus M_8)\bullet H^4 \oplus \ldots)\bullet H^4$
$\oplus (((M_1\bullet H^4 \oplus M_5)\bullet H^4 \oplus M_9)\bullet H^4 \oplus \ldots)\bullet H^3$
$\oplus (((M_2\bullet H^4 \oplus M_6)\bullet H^4 \oplus M_{10})\bullet H^4 \oplus \ldots)\bullet H^2$
$\oplus (((M_3\bullet H^4 \oplus M_7)\bullet H^4 \oplus M_{11})\bullet H^4 \oplus \ldots)\bullet H$

Each of the four elements is iterative (like in the equation above), but acting on every fourth message element independently of the other elements.

- Precompute H, H^2, H^3, and H^4
- Use H^4 during the vectorized state iterations
- Post-multiply the four final state elements by H, H^2, H^3, and H^4, respectively, and add the sub-totals to get the final result

- Vectorized iterations (2 of 4 shown):

$X0_{k+1} = (X0_k \oplus M_{4*k})\bullet H^4$

$X1_{k+1} = (X1_k \oplus M_{4*k+1})\bullet H^4$

(k = 0 .. (N-1)/4)
$X0_0, X1_0, X2_0, X3_0 = 0$

Iterate

Update ciphertext ***vct*** (see previous page)
vadd ***vtemp, vstate, vct***
vmulr ***vstate, vtemp, vh4, vredpoly***
(perform post-processing after last iteration)

AES-GCM Summary

- AES-GCM performance estimates and comparisons
  - Highly implementation dependent – the RISC-V estimates are rough-order-of-magnitude only
  - RISC-V RV32IVY[1] Assumptions:
    - 16 S-box AES vector functional unit (VFU), not pipelined. 16 clock cycles per 14-round (128-bit) encryption
    - Only one lane of wide-arithmetic w/ a (for example) 4-stage 64x64 pipelined core. VFU does four $\mathbb{GF}_2128$ multiplications (incl. reductions) every 20 clock cycles
    - Sixteen 128-bit (16B) elements per vector register (2048b=256B). Message size is 8Kb=1Kbytes=32 blks
    - Load/Store, AES, and $\mathbb{GF}_2128$ VFUs can operate in parallel

RISC-V RV32IVY	ARM Cortex™-M3[3]
2K clock cycles	192K clock cycles

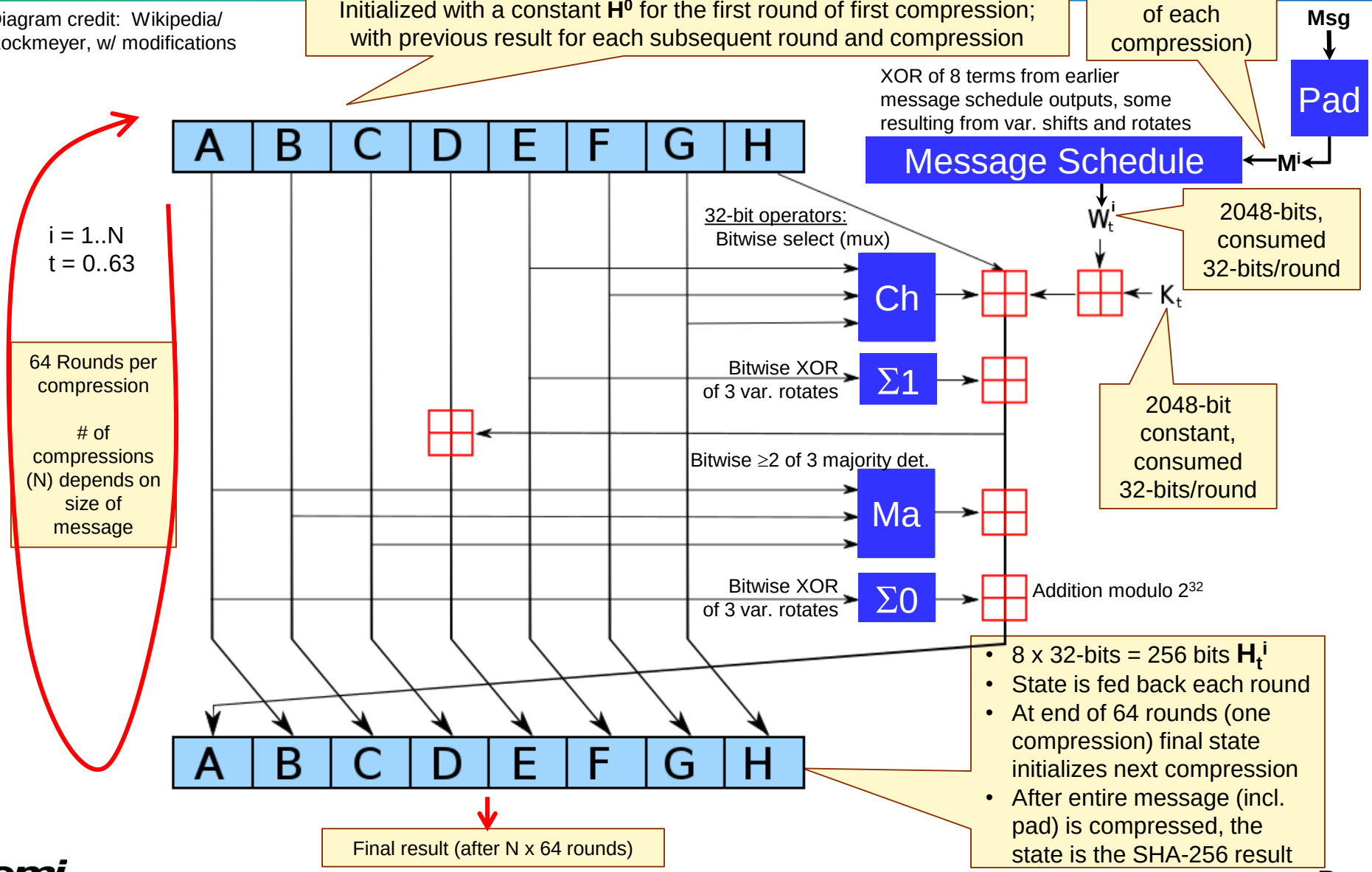
[1] Using letter "Y" for the RISC-V Cryptographic Extensions (not approved yet)
[2] https://software.intel.com/sites/default/files/m/4/1/2/2/c/1230-Carry-Less-Multiplication-and-The-GCM-Mode_WP_.pdf
[3] https://csrc.nist.gov/csrc/media/events/lightweight-cryptography-workshop-2015/documents/presentations/session7-vincent.pdf

SHA-256 Algorithm Summary

Diagram credit: Wikipedia/ Kockmeyer, w/ modifications

8 x 32-bits = 256 bit state H_t^{i-1}
Initialized with a constant H^0 for the first round of first compression; with previous result for each subsequent round and compression

512-bits of input message, consumed 32-bits/round (for 1st 16 rounds of each compression)



- 8 x 32-bits = 256 bits H_t^i
- State is fed back each round
- At end of 64 rounds (one compression) final state initializes next compression
- After entire message (incl. pad) is compressed, the state is the SHA-256 result

SHA-256 Vector Assembly Code Example

```
# SHA-256 using vector/crypto extensions: 4 x 512b of msg compressed per opcode
  vsetcfgd              SHA_CFG        # Config 3 vectors: 2reg w/ 4 elements x 64B (512b) bitstrings
                                       # and 1reg /w 4elements x 32B (256b) bitstrings
  vcrypt.init,SHA256    v3             # Select SHA-256 & initialize fixed constants
  vsetvl                t1, a1         # Set veclen temporarily to one element (a1=1)
  vcld                  v1, a3         # Load 512-bit that includes tail of msg
  vcrypt.pad.SHA256     v2, v1, t2     # Pad v1->v1,v2 starting @ bit t2 (SHA256 pad)
  vst                   v1, a3         # Store first padded 512-bit element
  bltu                  t2, t3, stripmine # If t2 < t3 (t3=512), v2 not needed for pad
    addi                a3, a3, 64     # Increment address for tail because of pad
    vst                 v2, a3         # Store extra 512 bits generated by padding
    addi                a0, a0, 1      # Increment N because of extra padding
stripmine:
    vsetvl              t0, a0         # a0 holds veclen N (#512b chunks);t0 ≤ MVL(=4)
    vld                 v1, a2         # Get first/next 4*512b (256B) of message
    vcrypt.hash.SHA256  v3, v1         # compress (4 compressions w/64 rounds each!)
    sll                 t1, t0, 6      # Multiply count x64 to get byte-delta
    add                 a1, t1         # Bump address pointers by byte-delta
    sub                 a0, t0         # subtract number of elements done from N
    bnez                a0, stripmine  # Loop if more message to process
  vstr                  v3, a5         # Save 256b (32B) digest result
```

Case Study: WalnutDSA™ Signature Verification

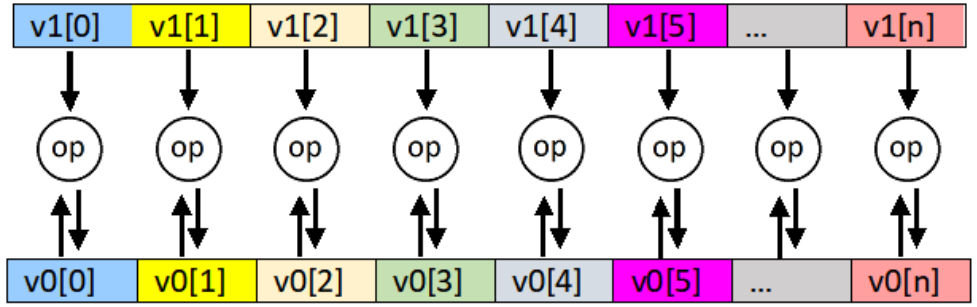
- SecureRF's Group-Theoretic-Cryptography-based digital signature algorithm
- Allows a signer with a fixed private/public key pair to create a digital signature associated to a given message which can be validated by anyone who knows the public key of the signer
- Main operation performed is "E-Multiplication", which primarily consists of matrix multiplication in the binary extension field $\mathbb{GF}_{2^5}$
- Multiplicand matrix in E-Multiplication is known as a *Colored-Burau* (CB) matrix
  - Derived from a braid generator b_i and a T-value (i.e. a given value in the finite field) t_i

$$CB(b_i) = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & t_i & -t_i & 1 & \\ & & & \ddots & \\ & & & & 1 \end{pmatrix}, CB(b_i^{-1}) = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & 1 & -\frac{1}{t_{i+1}} & \frac{1}{t_{i+1}} & \\ & & & \ddots & \\ & & & & 1 \end{pmatrix}$$

- Signature verification metrics presented are based on 128-bit security level (similar to ECC P256 and RSA3072) and are independent of the message hashing operation

Case Study: WalnutDSA Signature Verification

- Colored-Burau's similarity to the identity matrix allows for optimizations
- Given columns *a, b, c* of the multiplier matrix, where *a, b, c* correspond to *i-1, i, i+1* respectively, standard *n*-by-*n* matrix multiplication can be substituted by the following three equations:
  - $c_{k_{final}} = c_{k_{initial}} + b_{k_{initial}} \quad (for\ k = 1,2,\ldots,n)$
  - $b_{k_{final}} = b_{k_{initial}} * t_i \quad (for\ k = 1,2,\ldots,n)$
  - $a_{k_{final}} = a_{k_{initial}} + b_{k_{final}} \quad (for\ k = 1,2,\ldots,n)$



- Repeat for all braid generators
  - The product matrix from one iteration becomes the multiplier matrix to the next–this is an iterative operation
- Vector extensions allow for two sources of optimization
  - **(a)** Parallelize the three equations such that each column element *k* is operated simultaneously
  - **(b)** Use vector registers to limit the number of load/store instructions; and reduce instruction bandwidth, in general
- Implementation
  - Load each row of the column-major multiplier matrix into a unique vector register **(b)**
  - Extract the braid generator and corresponding T-value **(a)**
  - Based upon the braid generator, operate on the three vector registers as defined in the above three equations **(a)**
  - Store each row vector to the column-major multiplier matrix **(b)**
- Note that the first and last steps of the implementation are executed once; the remainder are looped for each braid generator

Case Study: WalnutDSA Signature Verification

Without cryptoextensions

SECTION	cycles per iteration	iterations	total cycles
exit	4	2	8
walnut_emul	1	2	2
loop	32	1704	54532
positive	5	1704	7668
negative	5	1704	7668
continue	106	1704	180624
		TOTAL	250502

With cryptoextensions

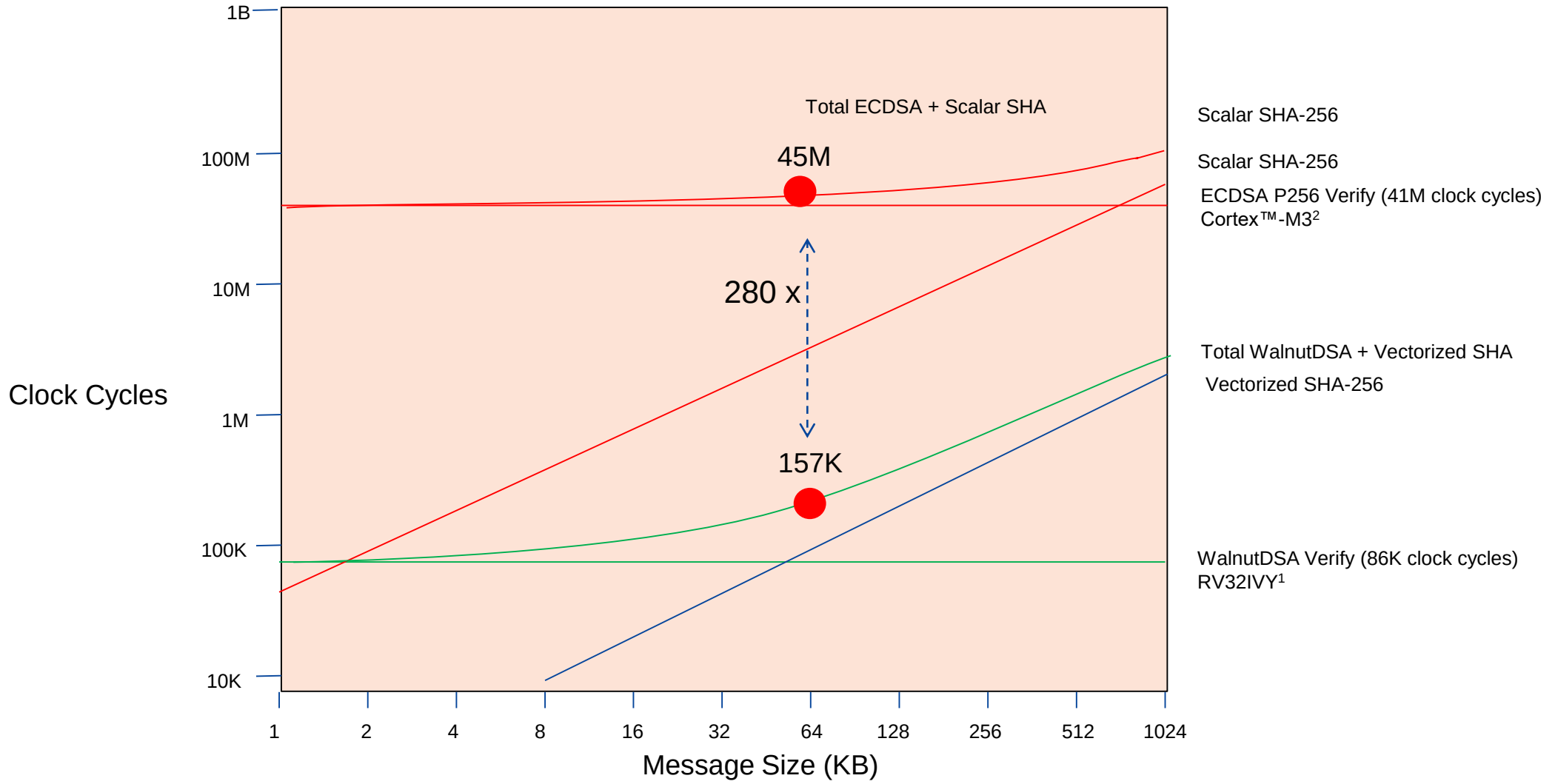
SECTION	cycles per iteration	iterations	total cycles
exit	18	2	36
walnut_emul	17	2	34
loop	30	1704	51124
positive	4	1704	6816
negative	4	1704	6816
continue	7	1704	11928
emultiply	5	1704	8520
		TOTAL	85274

2 functions calls due to two distinct braids: encoded message and signature

1704 braid generator estimate (1000 per signature + 704 per encoded message)

- Nearly a 3x speedup with crypto extensions vs. without crypto extensions
- Extra cycles in exit and walnut_emul are due to the loading and storing of the vectors registers, respectively
- With crypto extensions, continue is split into continue and emultiply. The latter is structured like a jump table, allowing for the easy selection and operation of the three vector registers without conditional branching.

Fast Secure Boot – Putting SHA-256 and WalnutDSA together



[1] Using letter "Y" for the RISC-V Cryptographic Extensions (not approved yet)

[2] https://csrc.nist.gov/csrc/media/events/lightweight-cryptography-workshop-2015/documents/presentations/session7-vincent.pdf